# Evaluating a Peer-to-peer Storage System in Presence of Malicious Peers

Samira Chaou, Gil Utard
*UbiStorage, 20 avenue Paul Claudel, 80000 Amiens, France*
*{samira.chaou,gil.utard}@ubistorage.com*

Franck Pommereau
*IBISC, University of Évry, 91000 Évry, France*
*franck.pommereau@ibisc.univ-evry.fr*

## ABSTRACT

*We present a peer-to-peer based storage system and evaluate its resistance in the presence of malicious peers. To do so, we resort to simulation of the actual code borrowed from the production system. Our analysis allows to identify the main threats, prioritise them and propose directions for mitigating the attacks.*

**KEYWORDS:** *Cryptographic protocols, security, storage, peer-to-peer, simulation.*

## 1. INTRODUCTION

Ubiquitous Storage (UbiStorage for short) develops and commercialises a storage solution based on a peer-to-peer network allowing each user to securely store and retrieve data [1], [2]. This system is based on a set of protocols allowing peers to exchange pieces of information in a fully distributed manner. The system is currently provided to UbiStorage consumers as a box that is ready to plug on the consumer's network and is pre-installed with a Linux system running the peer-to-peer software as well as end-user services. In [3], the security of these protocols has been assessed from a qualitative point of view, resorting to a formal modelling of the protocols combined with automated model-checking of typical scenarios as well as a manual proof. Model-checking allowed to discover potential flaws based on replay attacks, fixes have been proposed and their quality checked again using model-checking. A manual proof allowed to show that no intruder external to the network (*i.e.*, whose public key is not recognised by peers) could ever learn any piece of data stored by the peer-to-peer network.

In this paper, we would like to complete further this initial work by assessing the resistance of the peer-to-peer network in the presence of malicious peers, *i.e.*, attackers that are part of the network. This cannot be analysed from a qualitative point of view because in such conditions, there obviously exists attacks and modelling is not needed to discover them. More interesting is the quantitative point of view, which is the topic of this paper. Using various simulations, we will show that the system can resist to various attacks as long as the number of malicious peers remains limited. Such attacks would require that a number of users gain control over their boxes and coordinate their actions to perform the attack, which is unlikely to occur in practise. Knowing how many internal attackers the system can resist to is important for UbiStorage because this can be partially controlled, for instance by avoiding to allocate too many boxes to a single customer.

The rest of the paper starts with a presentation of the peer-to-peer system architecture. Then, section 3 presents the attack scenarios we have considered. How they are implemented as simulations is then described in section 4. Next, the results from these simulations are presented in section 5. Concluding remarks are finally given in section 6.

## 2. THE UBISTORAGE SYSTEM

The system developed by UbiStorage is fully distributed using peer-to-peer communication. Each peer in the system is a network node that corresponds physically to a box allocated to a given customer, and

runs the various services and clients composing the system. The system is structured in three layers:

- the *application layer* is directly queried from the end-user interface and is composed of client processes to perform three basic primitives: primitive *Put* is used to store a file in the system, primitive *Get* to get back a stored file, and primitive *Delete* to remove a file from the system;
- the *communication layer* consists of a distributed hash table (DHT) that stores all the information needed by the application layer;
- the *routing layer* uses a key-based routing protocol to dispatch the messages exchanged within the DHT.

Each peer is identified by a unique 128 bits identifier called *PeerID* and is the responsible for a range of file identifiers (128 bits *FileID*s) and is the owner of another range of identifiers: when a peer needs to store a new file, it must choose an unallocated FileID it owns and send a storage request to the peer that is responsible for the chosen FileID.

## 2.1. Application layer

The *Put* primitive is called when a file needs to be stored on the system. To do so, the file is first fragmented using a Reed-Solomon error-correction code [4]. Compared with a simple duplication, this allows to reduce the amount of storage required and consequently the amount of network communication. The fragments resulting from this phase constitute the basic data unit that is exchanged between the peers. Reed-Solomon code is parametrised with two positive integers $s$ and $r$, resulting in building $s + r$ fragments for each file, and defined in such a way that any set of only $s$ fragments is required to reconstruct the corresponding file. Each stored file is thus given a unique identifier that is also the identifier of each fragment of this file. The information about the location of the fragments for each file, so called *file meta-information* or *FileMI*, is maintained within the network in order to be able to retrieve files, either to reconstruct a file requested by a user when primitive *Get* is invoked, or to ensure that enough fragments to reconstruct any file always remain available. The latter aspect requires a constant monitoring to detect the possible disappearing of peers and trigger the reconstruction and redistribution of files before they become unavailable.

Each peer is internally organised as depicted in figure 1 to provide two main features: management of fragments and of meta-information, both using the hard-disk. More precisely:

- the *client service* is responsible for executing the Put, Get and Delete primitives in response to requests from the end-user;
- the *storage service* (StrS) receives fragments to be stored from other peers or sends them back as requested;
- the *meta-information service* (MIS) stores all the meta-information related to the fragments locally stored. Each MIS is responsible for a range of FileIDs and associated to a *leaf-set* composed of the peer's neighbours in the network. Notice that each peer belongs to several leaf-sets;
- the *monitoring service* (MonS) monitors the MIS and issues warnings when critical files are found, *i.e.*, files for which there exist just enough fragment to allow reconstruction;
- the *reconstruction service* (RecS) is responsible for actually reconstructing files and redistribute fragments within the leaf-set when an alert is sent by the monitoring service.

Getting a file is executed in two phases: first get the corresponding meta-information, called the *FileMI* and that includes a *FileID*, which allows to know all the storers that save a fragment of this file; then send to these storers a request to get fragments, which allows to reconstruct the file. Similarly, putting a file is also executed in two phases: first get a list of storers which are available to store fragments of the file; then, send requests to the selected storage services to store the fragments. The list of storers is requested to the MI service that is currently online and has the PeerID the closest to the chosen FileID (see routing below). The returned list of storer services is chosen among those in the leaf-set of the contacted MI service. Deleting a file is performed similarly to getting it, except that the storers will forget everything about the fragments instead of sending them.

## 2.2. Communication and Routing Layers

Communication in the system is based on the distributed hash table (DHT) Pastry [5]. Network nodes are logically organised as a ring and identified by their PeerIDs. To facilitate communication between peers, several tables are used, in particular, each peer maintains three tables:

- *routing table*: each time a peer communicates with another, it adds its IP address and PeerID to the routing table;
- *leaf-set table*: a neighbour table is maintained by

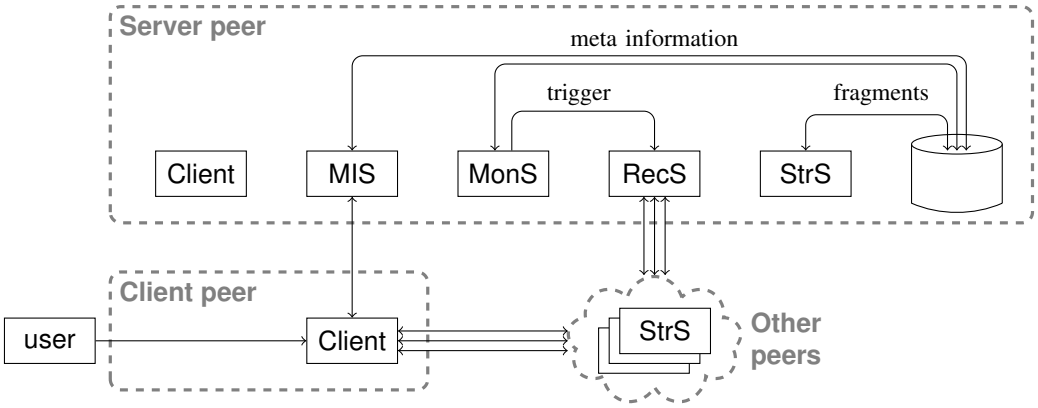



**Figure 1. Organisation of Each Peer and of its Communication with Others. Each client communicates directly with the MI service (MIS) that manages the file involved in a Put, Get or Delete, and possibly with the storage services (StrS) that hold fragments of this file. The monitoring service (MonS) triggers the reconstruction service (RecS) that in turn communicates with the storage services holding the required fragments. The hard disk is used to store both meta-information and file fragments.**

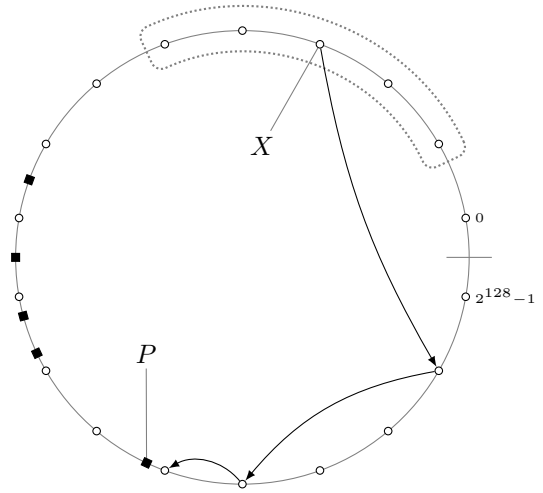



**Figure 2. Organisation of the Distributed Hash Table. White dots denote peers and black squares denote files. The dotted region corresponds to the leaf-set of $X$.**

each peer, it is updated regularly and evolves with the arrival and departure of peers;

- *jump table*: pairs whose PeerID are located at distances $2^k$ are stored as a secondary routing table. This table allows to find faster routes across the DHT.

How these tables are used is illustrated in figure 2. All the possible 128 bits PeerIDs are considered as logical addresses within a ring (*i.e.*, working modulo $2^{128}$). Each peer is located at a given PeerID depicted by a white dot. FileIDs are considered similarly and depicted as black squares. Consider for instance the node tagged by $X$. Its leaf-set is depicted as a dotted region and contains all its neighbour peers whose IP addresses are maintained in the leaf-set table. Assume now that $X$ needs to send a message to a PeerID at position $P$, for instance to request a list of storers for a new file whose FileID is also $P$. In the first case (not depicted), if $P$ is within the leaf-set of $X$, messages are send directly using the leaf-set table. Otherwise, as depicted in figure 2, the jump and routing tables will be used to route the message throughout the DHT: $X$ first sends the message to the peer it knows that is the closest one to $P$ (first arrows outgoing from $X$); then, such intermediate peers will forward the message using the same principle; eventually, the message will arrive to the peer that is the closest one to $P$ among those that are presently online, which is known because this last peer knows all of its leaf-set. The routing table thus acts as a cache, while the jump table allows to accelerate routing by providing shortcuts through the ring of the DHT.

## 3. INTRODUCING MALICIOUS PEERS

In this section we consider several cases of malicious peers that could be introduced in the system, for instance if a customers gains control over a box. Then we discuss how such modified peers could influence the global behaviour of the system and the service provided to other peers. We can consider five cases:

- malicious storage services lying about the outcome of requests or trying to recover information from stored fragments;
- malicious MI services sending incorrect FileMIs;
- malicious reconstruction services that do not launch reconstruction when critical files are found;
- reconstruction of files by malicious peers trying to gather enough fragments to do so;
- monitoring services preventing reconstruction to actually occur;

For our simulations we retain only the three first cases, and ignore the two remaining ones because there results are already predictable without the need for simulation. Indeed, a malicious monitoring service is equivalent to a malicious reconstruction service: reconstruction does not occur. Concerning illegitimate reconstructions of files, we observed that it can occur only in case of a small network (which is far from real case) because otherwise, fragments are so much distributed that it is statistically difficult to obtain enough of them to attempt a reconstruction. Moreover, the fragmented files are ciphered using strong cryptography which makes a reconstruction useless. As discussed below, the main threat is the loss of files in the three considered cases.

## 3.1. Malicious Storage Services

Two malicious behaviours can be envisaged for a storage service: pretend it stores a fragment while it does not, or pretend it does not store a fragment while its does. In the first case, the storer may as well actually save the fragment but then refuse to send it back upon a later request (or send an invalid fragment). So, in both cases, the storer may try to accumulate enough fragments of a given file in order to allow a reconstruction. The second behaviour (store but pretend not to) is probably a better strategy for a malicious storage service in order to attempt a reconstruction: indeed, the rest of the system, and in particular a client willing to put a file, may try to avoid sending more than one fragment of a file to each storer; by faking failures, a storer may actually accumulate more fragments than what others in the system can assume. The possibility of such malicious reconstruction has been actually observed on simulations, but fortunately, as discussed above, there exists a simple fix that consists in encrypting files before to store them. So we concentrate here on another threat that is file losses, which turns out to be the main concern for customers when the problem of privacy has been solved.

**3.1.1. Malicious Success on Store (Attack A1).** Obviously, this first malicious behaviour can result in file losses. When asked to store a fragment, the storage service pretends it succeeded but actually the fragment is discarded, or it is stored but will never be sent back when requested, or, equivalently, an invalid fragment is returned instead (which is more difficult to detect as a malicious behaviour—we have implemented this case). If at least $r + 1$ storers lie this way for a given file, it will never be possible to reconstruct the file because less than $s$ of the required fragments to do so actually remain in the system.

It is quite difficult to protect the system against such an attack, in particular if it occurs during a reconstruction triggered by a monitoring service. Indeed, we can imagine that on a Put, the client may hold the file some time and, before to report a success, it may challenge the appropriate storage services to check that the fragments are actually stored. But after Put has succeeded, the file is assumed to be safely stored on the system and is likely to be deleted from the user's computer. So, the client may attach to the FileMI a series of precomputed challenges that could be used by the MIS to check that fragments are effectively stored. A challenge can consist of a random value and a hash of each fragment concatenated with this value. A storer can then be sent the random value and challenged to compute the hash from its fragment. Each challenge does not cost too much storage and a series of challenges can be easily computed during the Put. Then, at each reconstruction, more challenges can be generated again to replace the used ones. The main drawback of this method is that it requires additional operations in the system, but fortunately these are computational tasks with very few network communication. Moreover, correctly calibrating how often storage services are challenged is important. Indeed, when all the challenges of a series have been used, it may be necessary to reconstruct the file in order to generate new challenges, which results in a large amount of network communication. Knowing how the system is resistant to attack A1 is the first step to perform a correct calibration of this counter-measure.

**3.1.2. Malicious Fail on Store (Attack A2).** When asked to store a fragment, a malicious storage service may pretend to fail. As a consequence, it will never be asked for this fragment since it is not assumed to hold it. If enough storage services behave this way, a Put may fail, which is not a security problem. But also a reconstruction may fail, resulting in a file loss, which is now a more serious concern. Notice that this

is quite a different situation compared with the case of real failures: this latter case may occur in particular when a disk is full, but the overall available storage space can be monitored from a global point of view and increased by plugging more boxes into the system or adding bigger disks to the existing boxes. So, this turns out to be more a maintaining concern. But in the presence of malicious storage services, the overall available space may be sufficient to guarantee the possible reconstruction of a file, but the faked failures may anyway result into a file loss.

Like the previous malicious behaviour, this one is difficult to combat because it can be detected only when it is too late. So, in both cases, it is important to measure the resistance of the system against malicious storage services.

### 3.2. Malicious MI Services (Attack B)

A MI service is responsible for storing and maintaining the list of storers that hold the fragments of a given file. A malicious MI service may intentionally destroy or corrupt this information, either when it stores it or when it sends it back on request. We have implemented a malicious MI service that sends invalid FileMIs when requested, which is probably the most difficult situation to detect as a malicious behaviour. This of course immediately results in file losses because the data to recover the file is actually stored in the system but it is not known where.

This is somehow a less critical problem than the previous one because meta-information does not represent a large amount of data and can be easily replicated. Actually, this is already the case in some way because an exploration of the system can be attempted to find lost fragments and rebuild meta-information. But this countermeasure has a cost that needs to be considered with a clear understanding of the resistance of the system against malicious MI services.

### 3.3. Malicious Monitor or Reconstruction Service (Attack C)

When too many peers holding fragments of a given file go offline or crash, it is necessary to reconstruct the file and redistribute its fragments before there is not enough available fragments to do so. The monitor service is responsible for detecting this situation and triggering the reconstruction service to actually perform the required tasks. If one of these services is malicious and does not actually fulfill its duty, files will be inevitably lost.

To combat such attacks, monitoring and reconstruction can be duplicated, *i.e.*, several peers in a leaf-set may be simultaneously responsible for performing these tasks. But as in the previous case, this induces an extra cost and also it requires an additional coordination between the redundant services, which increases the complexity of the system.

## 4. SIMULATION ARCHITECTURE

To assess the resistance of UbiStorage system against malicious peers as defined above, we can resort to simulation. This allows in particular to measure how many files can be lost with respect to the number of malicious peers in the network.

We have designed and implemented a fully configurable simulator that embeds the actual code executed on the boxes. The simulator takes as input a configuration file that describes the simulation environment, *i.e.*, information about: the number of peers in the simulated network; the number of malicious peers in the system; the number of files stored in the system; the duration of the simulation; the reconstruction threshold (*i.e.*, the minimal number of fragments after which a reconstruction must be undertaken); the leaf-set size.

Taking this information in account, a trace of events are generated, it is called a *churn trace* and describes the departure and arrival of peers in the system. Three events are considered: creation of a new peer (*i.e.*, installation of a new box), definitive disappearing of a peer (*i.e.*, crash of a box), and transient disappearing of a peer (*i.e.*, reboot of a box, shut down during the night, or temporary network failure). The simulation engine then follows the churn trace, which triggers reactions of the peers as in the real system. Notice that we have not considered clients in the simulation but instead we have started from a system that already stores file. Indeed, as explained in the previous section, our primary goal is to assess the resistance of the storage system in terms of file losses.

During the simulation a state manager monitors the system and gives information about the state of the system. In particular, it reports the number of files still stored in the system (*i.e.*, that can be reconstructed), and the number of dead files (*i.e.*, for which the meta-information has disappeared or there does not exist enough fragments to allow reconstruction). We have run a number of simulations with various initial setting

*Attack A1: malicious success on store*

| % malicious | $\leq 5$ | 6 | 7 | 10 | 15 | 20 |
|---|---|---|---|---|---|---|
| % dead files | 0 | 0 | 0 | 0.67 | 1.29 | 3.64 |

*Attack A2: malicious fail on store*

| % malicious | $\leq 5$ | 6 | 7 | 10 | 15 | 20 |
|---|---|---|---|---|---|---|
| % dead files | 0 | 0 | 0 | 0 | 1.71 | 5.35 |

*Attack B: malicious MI service*

| % malicious | 0 | 1 | 2 | 3 | 5 | 7 |
|---|---|---|---|---|---|---|
| % dead files | 0 | 1.57 | 3.92 | 4.08 | 4.51 | 5.71 |

*Attack C: malicious monitor/reconstruction service*

| % malicious | 0 | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|---|
| % dead files | 0 | 1.66 | 2.01 | 2.60 | 4.26 | 5.94 |

**Figure 3. Simulation Results for the Considered Attacks.**

of the parameters, in particular of the percentage of malicious peers of each kind.

## 5. EXPERIMENTAL RESULTS

In this section, we report the observations draw from a series of simulations for a network of 200 peers storing 10 000 files whose fragments are uniformly distributed, with a reconstruction threshold of 50% (*i.e.*, if $r/2$ fragments are no more available, a reconstruction is attempted). This parameter can be adjusted to easily improve the resistance to attacks but at the same times, it leads to an increased amount of costly operations. The chosen leaf-set size is 16 nodes and the simulation duration has been set to 1 year and 23 days. In average, each simulation has run for about 75 minutes. These choice of the parameters allows a reasonable simulation time while already demonstrating interesting behaviours. We have also ran simulations with higher numbers or peers and files, resulting in similar results. This also allowed to check that, for a given number of files, the number of files lost decreases with the number of peers: the system is more resistant when each peer is in charge of less files.

Figure 3 and figure 4 summarises the results of these simulations. (Attacks A1, A2, B and C are named consistently with respect to the corresponding paragraphs in section 3.)

As one can see in figure 4, attacks B and C are much sooner efficient than attacks A1 and A2. This is not surprising because the latter threaten fragments while the former directly threaten files, either at the level of meta-information (attack B) or by letting them disappear with peers (attack C). As explained above, attack B can be mitigated by maintaining multiple copies of the meta-information. Similarly, attack C can be mitigated by duplicating the monitoring and reconstruction process, which also requires to duplicate meta-information (each monitoring service needs the FileMIs for all the files it monitors). So both questions are clearly related and rely on a consistent duplication of meta-information. Fortunately, part of this problem has been solved already because it is a crucial requirement in a network where peers can be disconnect at any time (and reconnected later): even if a peer that is the serves a FileMI goes offline, the corresponding file should be recoverable without waiting until the peer goes online again (which will not occur if the peer has crashed). To allow this, we exploit the fact that FileMIs can be recovered from the peers within a leaf-set that store fragments for the corresponding file. When a peer goes offline, the meta information it served has to be now served by other peers in its leaf-set; conversely, when a peer goes online, it becomes responsible for meta information it may not know (for instance for files Put when it was offline). So, in this situations, FileMIs are reconstructed using information attached to fragments and the newly online MI service collects it within its leaf-set to reconstructs the needed FileMIs. So, attacks B and C can be mitigated by exploiting this mechanism in order to search an invalid or missing FileMI on neighbour peers of the malicious one. However, this leads to an agreement problem that is known to be difficult in distributed systems [6]. Moreover, the underlying routing does not allow to choose which peer is contacted during a communication: by construction of a DHT only the online node with the ID the closest to the requested ID can be contacted. Changing this looks like a very complex extension that would be better to avoid.

It is also interesting to note that UbiStorage system exists in two versions. One is fully distributed and has been studied in this paper, an earlier version is based on a central MI service and has been considered in [3]. The sensitivity to attacks B and C is an argument in favour of the centralised version. But this can be reversed by considering that all the meta-information for the whole system can be lost simultaneously in case of an attack of the central server (that becomes a single point of failure). So we believe that the distributed version considered in this paper is a better basis to build a secure system.

A possible compromise is to introduce in the distributed system a centralised MI service to serve as a fallback in the case of a failure of the MI services located on the peers. Ideally, such a central MI service

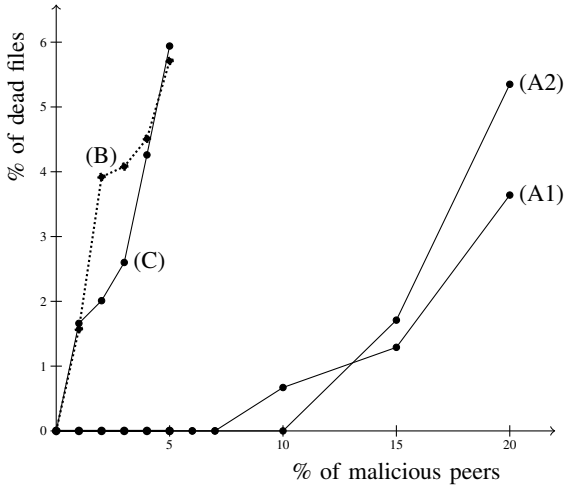



**Figure 4. Dead Files with Respect to the Percentage of Malicious Peers.**

could be in the cloud, improving its reactivity and its resistance to attacks. Notice that this solution allows to find a simple solution for the agreement problem as long as the central MI service is considered reliable.

Concerning attacks A1 and A2, they can be seen as actual tests of the robustness of the system. And we can observe a satisfactory resistance of the system, which can for sure be improved again by increasing the redundancy of fragments (*i.e.*, considering a larger value of the parameter $r$). The resistance of the system can also be improved by increasing the critical threshold of the reconstruction, fixed to 50% in our simulations. Other simulations done with a higher critical threshold showed that the lost of files indeed decreases when the threshold increases. Somehow, this kind of attacks is not fundamentally different from simple peer failures (*i.e.*, fragments are lost) and the system is exactly designed to resist to this problem. So, we can assume that a correct maintaining and monitoring of the network of peers can efficiently prevent the multiplication of malicious peers and that they may be detected using appropriate challenges before file losses actually occur.

# 6. CONCLUSION

We have examined the peer-to-peer based storage system developed by UbiStorage and how its resistance to malicious peers can be analysed using simulation. It turned out that the main threat is the loss of files stored in the system by users. When malicious peers only attack the stored data, we have seen that the resistance of the system to peers disappearing provides also a good resistance to such attacks. However, we have seen also that attacks against meta-information is a much more critical problem. We have proposed directions to mitigate this issue by exploiting the existing redundancy of the meta-information and possibly introducing a centralised meta-information server that should be preferably implemented as a cloud service.

Future work will address the evaluation of combined attacks and the design of defence strategies that will be evaluated again using the simulation techniques presented in this paper. We will also address the crucial question of the detection and exclusion of malicious peers from observations of their behaviours (and using a reputation system like in [7], [8]), which should greatly help to avoid relying on them. Another direction for simulation is to evaluate performances and we are currently working on the questions of measuring reconstruction times, bandwidth usage, put and get times, input/output amount, etc.

## 6.1. Related works

Two main types of distributed network storage can be compared: centralised storage as a cloud service and decentralised one as a peer-to-peer service. Cloud-based systems are likely to be less complex than P2P system and can be seen as a generalisation of client/server technologies combined with balancing mechanisms. Most of the complexity resides in maintaining the consistency of a redundant distributed information. These problems also occur with P2P systems together with additional problems like agreement, addressing and routing as discussed above. On the side of infrastructure, P2P system are much lighter than cloud-based systems. By exploiting comparatively smaller computation and storage units distributed over the network, the overall system is likely to be less expensive and to consume less energy. At the same time, P2P system are likely to be more robust and scalable than cloud-based systems because they do not rely on any centralisation. However, this centralisation is a serious advantage when it comes to security: a cloud infrastructure is entirely controlled by one operator and can be globally secured while a P2P system is only loosely controlled. Problems related to data confidentiality can be solved easily using cryptographic tools. But as shown in this paper, integrity and availability problems are much more complex to address in the presence of potentially malicious peers.

As suggested above, there should exist a compromise combining a P2P network to perform most of the

required operations together with cloud-based services to address security issues related to meta-information, acting both as an arbiter and a fallback. Such a combination is likely to exploit the best of both worlds by displacing most on the charge toward the P2P network.

Performance evaluation is quite a classical use of simulation in the domain of peer-to-peer systems, see for instance [9], [10], [11]. However, to the best of our knowledge, assessing security using simulation is not as common. In [12], experimental results are presented in case of attacks against the certification system used to detect and exclude misbehaving nodes. This corresponds to an attacker that is external to the system, while we consider attackers from within the system. The certification system in [12] is based on the principle of consensus decision making, and the simulation shows the resistance of the system to false detections and exclusions (*i.e.*, a honest node is detected as misbehaving and is possibly banned from the network). In [10], simulation is used to measure the average time taken by a node to join the system. Various combinations of two systems (Secure Spread and Gnutella), two cryptographic algorithms (RSA and DSA), and two methods (static and dynamic threshold) are considered. In [11], simulation is used to measure the availability, reliability and resource usage: it shows how availability increases with the degree of duplication of files in the system, and how reliability can be ensured by cryptography and versioning. In [9], simulation is performed on the Pond OceanStore prototype to analyse system performance: it shows how storage overhead increases with the size of stored data, an update performance, archive retrieval performance and replication costs are analysed.

## ACKNOWLEDGMENT

The authors would like to acknowledge Sébastien Choplin and Hung-Cuong Le for their decisive participation in the design and implementation of the simulator used in this paper.

## REFERENCES

[1] Ubiquitous Storage company. [Online]. Available: `http://www.ubistorage.com`

[2] SPREADS project. [Online]. Available: `http://www.spreads.fr`

[3] S. Sanjabi and F. Pommereau, "Modelling, verification, and formal analysis of security properties in a P2P system," in *Workshop on Collaboration and Security (COLSEC'10)*, ser. IEEE Digital Library. IEEE, 2010.

[4] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *SIAM journal on applied mathematics*, vol. 8, no. 2, 1960.

[5] R. Mahajan, M. Castro, and A. Rowstron, "Controlling the cost of reliability in peer-to-peer overlays," in *IPTPS'03*, 2003.

[6] L. Lamport, R. Shostak, and M. Pease, "The Byzantine generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, 1982.

[7] M. Morvan and S. Sené, "A distributed trust diffusion protocol for ad hoc networks," in *ICWMC Second International Conference on Wireless and Mobile Communications*, 2006, p. 87.

[8] T. Cholez, I. Chrisment, and O. Festor, "A distributed and adaptive revocation mechanism for P2P networks," in *Proceedings of the 7th International Conference on Networking*. IEEE Computer Society, 2008.

[9] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz, "Pond: the OceanStore prototype," in *2nd USENIX Conference on File and Storage Technologies (FAST'03)*, 2003.

[10] N. Saxena, G. Tsudik, and J. H. Yi, "Admission control in peer-to-peer: Design and performance evaluation," in *ACM Workshop on Security of Ad Hoc and Sensor Networks*, 2003.

[11] C. Batten, K. Barr, A. Saraf, and S. Trepetin, "Pstore a secure P2P backup system," Projet Pstore, 2001.

[12] F. Lesueur, L. Mé, and V. Viet Triem Tong, "Detecting and excluding misbehaving nodes in a P2P network," *Studia Informatica Universalis*, vol. 7, no. 1, 2009.